Delft University of Technology
Software Engineering Research Group
Technical Report Series

Declaratively Programming the Mobile Web with Mobl

Z. Hemel, E. Visser

Report TUD-SERG-2011-001

TUD-SERG-2011-001

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Declaratively Programming the Mobile Web with Mobl

Zef Hemel
Software Engineering Research Group
Delft University of Technology, The Netherlands
z.hemel@tudelft.nl

Eelco Visser
Software Engineering Research Group
Delft University of Technology, The Netherlands
visser@acm.org

ABSTRACT

A new generation of mobile touch devices, such as the iPhone, Android and iPad, are equipped with powerful, modern browsers. However, regular websites are not optimized for the specific features and constraints of these devices, such as limited screen estate, unreliable Internet access, touch-based interaction patterns, and features such as GPS. While recent advances in web technology enable web developers to build web applications that take advantage of the unique properties of mobile devices, developing such applications is not a clean, well-integrated experience. Developers are required to use many loosely coupled languages with limited tool support and application code is often verbose and imperative. We introduce *mobl*, a new language designed to declaratively construct mobile web applications. Mobl integrates languages for user interface design, data modeling and querying, scripting and web services into a single, unified language that is flexible, expressive, enables early detection of errors, and has good IDE support. We illustrate the design of the language with the implementation of ConfPlan, an application for keeping track of the schedule of conference events.

General Terms

Mobile Web Development, User Interaction

Keywords

mobile web, web development, reactive programming, declarative programming

1. INTRODUCTION

With the rapid growth in sales of modern smart phones and tablets, such as iPhone, iPad, Android and BlackBerries, the web becomes available on an increasing number of powerful mobile devices equipped with modern web browsers. However, today's websites are optimized for personal computer browsers and environments, whereas mobile devices are used in different contexts, and have different features and constraints than personal computers, for instance:

- Internet access is not always available, reliable or fast;
- Screen estate is limited;
- Expected user interaction patterns are different, such as touch controls and gestures such as tapping, swiping and pinching;
- Applications are expected to respond to changes in context, such as holding the device in portrait or landscape mode, or changes in location.

Consequently, hundreds of thousands of custom *native* mobile applications are being developed. Examples include communication applications (e-mail, messaging), content viewers (books, articles, papers, RSS feeds, video, photos, audio) and location-based services (wikihood, foursquare, loopt). While these applications run locally on the device itself, a large class of these applications are *data-driven applications* that communicate with one or more web services to exchange data.

While iOS, Android, BlackBerry, WebOS, Windows Phone 7 and other platforms are similar in terms of interaction, features and restrictions, their development environments are quite different. iPhone and iPad applications are developed using the Objective-C language; Android and BlackBerry applications are built using Java, but using very different APIs; WebOS applications use HTML, CSS and JavaScript; Windows Phone 7 development is done using .NET. Developing software that is *portable* to multiple platforms is difficult. In addition, *deployment* is non-trivial; most platforms come with an application marketplace, some of which require manual testing of submitted applications by the marketplace provider before being published — a process that can take many weeks — and applications can be rejected for seemingly arbitrary reasons.

At the end of the 1990s, mobile phones started to gain access to the Internet through WAP (Wireless Application Protocol). The development model for WAP applications was very similar to the development of regular web applications. Rather than sending HTML, a server would send WML (Wireless Markup Language) to the mobile phone. With the release of the original iPhone in 2007, a new generation of smart phones and tablets started to be released with more powerful browsers that support all modern web technologies. At the same time, advancements in HTML (HTML 5) and CSS (CSS 3) started to enable the creation

of web applications that offer a comparable experience to native applications, especially for *data-driven applications*, by supporting application and data caching, detection of touch gestures and access to geographical position information (GPS). The portability and deployment advantages of web applications make the use of web technologies for building mobile applications very attractive.

Similar to *native* applications, mobile web applications can now be developed that run completely disconnected from the server, requiring a different development model than regular web applications. When the mobile web application is first launched through the web browser, its application code is cached on the device. The application can use local SQL databases to cache data obtained from a server for offline use. When no Internet connection is available, the mobile browser retrieves the application from its cache and continues to operate. All application logic, written in JavaScript, resides on the device rather than on the server as is the case in regular web applications. Communication with the server, similar to native applications, happens by performing web service calls using AJAX (Asynchronous JavaScript and XML). At the time of writing, HTML5 is well supported by the iPhone, iPad, Android and WebOS platforms, support is forthcoming for Blackberry.

While it is now *possible* to develop offline-capable mobile web applications that are portable and easy to deploy, developing such applications is not a consistent and well-integrated experience. We identify the following problems:

1. Development involves a mix of specialized, loosely coupled languages, such as HTML for creating user interfaces, CSS for styling, JavaScript for application logic, SQL for database querying and caching manifests for application caching. Inconsistencies between the languages are detected only at runtime. Earlier detection of faults would speed up development.
2. Tool support is sub-optimal. Due to the loose coupling of web languages, very little cross-language support is offered by IDEs (Integerated Development Environments). The dynamic nature of web languages prevents accurate implementation of desirable IDE features such as in-line error reporting, code completion and reference resolving.
3. While HTML enables declarative definition of user interfaces, it lacks abstraction mechanisms, e.g. to parameterize and reuse HTML fragments.
4. Establishing a connection between user interface and data typically requires a lot of boilerplate code for copying values from data objects to user interface and vice versa.
5. Many JavaScript APIs are defined as asynchronous APIs, forcing developers to write code in continuation-passing style.

In this paper, we introduce *mobl*[1], a high-level, declarative language for programming mobile web applications, which addresses these problems. Mobl integrates languages for user interface design, data modeling and querying, scripting and web services into a single, unified language. The language is *high-level* since it avoids accidental complexity such as continuation passing style and supports the definition of reusable user interface elements. The language is *declarative* since it ensures automatic updates of the user interface through reactive programming, automatic persistence of data in the client-side database, and automatic conversion of data from services. The integration of the various concerns of mobile web programming into a single language, enables consistency checking accross concern boundaries, ensuring early detection of many common errors by the mobl IDE (integrated into Eclipse), which provides in-line error reporting, code completion and reference resolving. The mobl compiler compiles mobl code into a combination of HTML, CSS, JavaScript and application caching manifests instructing the browser to cache the application locally.

We used mobl to create a number of mobile applications, ranging from simple task managers and tip calculators to a twitter client and simple graphical games. To illustrate the design of mobl, we describe the implementation of ConfPlan[2], a mobile application for viewing the schedule of conference events the user is attending. It uses GPS to find nearby conferences and downloads and caches their schedules transparently. The user can browse the schedule, search for keywords, read abstracts and mark talks as favorites to more easily decide what sessions to attend. ConfPlan is representative of a large class of mobile applications that present data obtained from a web service and use contextual information such as geographical location.

The rest of this paper is organized as follows. In Section 2 we analyze the architecture and interaction patterns of mobile web applications, and identify a number of problems related to constructing such applications. Section 3 introduces mobl, our solution to these problems, by demonstrating how it is used to build the ConfPlan application. Section 4 discusses the language design decisions made for mobl. Section 5 discusses the limitations of our approach and related work. In Section 6 we draw some conclusions.

2. MOBILE WEB APPLICATIONS

The design of a new language for mobile web application development requires a thorough understanding of the mobile domain. This section discusses the architecture of traditional web applications and compares it to the architecture of *mobile* web applications. Subsequently, we identify a number of problems in the *development* of mobile web applications.

2.1 Architecture

Regular web applications respond to HTTP requests from the client. When a request comes in, it is handled by a server written using, for instance, Java, .NET, PHP or Ruby. The server communicates with a database to retrieve or manipulate data, and eventually sends back HTML to the browser which renders it on the user's screen. A server handles multiple users and typically stores data for all its users in a shared database. HTTP requests can also be sent by JavaScript

[1] `http://www.mobl-lang.org`

[2] `http://researchr.org/confplan/confplan.html`

code on the web page, using AJAX calls. Based on the result of such a request, the JavaScript may manipulate the HTML DOM (Document Object Model) to make changes to the user interface without requiring an entire page reload. In addition to performing AJAX calls, JavaScript can also be used for client-side validation of user input in forms.

There are multiple approaches to developing *mobile* web applications. For older, non-smart phones, processing power is the main bottleneck. Therefore, several thin-client approaches exist [9, 8] where all processing happens on the server and phones are served with pre-rendered pages. However, today's modern smart phones have more powerful processors, thus client-side processing is no longer a bottleneck. Therefore, for these devices applications can be developed in a range of styles. On one end of the scale are web applications that are built similarly to regular web applications, except reducing the amount of data presented on a single page, to fit the screen size of the mobile device. It is relatively easy to adjust a regular web application to produce pages that are more friendly to the smaller screen size of a mobile device. A drawback of this approach is that such applications are not available without an Internet connection. In addition, Internet speeds on mobile devices are on average a lot slower than on PCs, resulting in a bad user experience.

At the other end of the spectrum are *offline-capable* mobile web applications that, once accessed by the mobile browser, are cached locally. They may fetch data from the server and cache it in a local database on the device as well. The development model of this type of application is very similar to desktop applications and native mobile applications. All the application logic, written in JavaScript, executes at the client, in the device's browser. Like desktop applications, such mobile web applications are single-user applications that do not require user authentication and access control. An additional advantage of this approach is that applications can be used without an Internet connection after the application and its data is loaded and cached locally. Internet latency on mobile networks is also less problematic because fewer requests have to be sent to the server.

In this paper we focus on mobile web applications that fully operate within the mobile web browser and are off-line capable. Moreover, we focus on data-driven applications, whose main purpose is browsing, searching and manipulating (textual) data. Thus, we do not consider applications such as graphical games, graphics editors, and applications for audio and video processing.

2.2 Mobile Web Development

While modern web technologies make development of offline capable data-driven web applications possible, development of such applications is not a well-integrated consistent experience. Web applications are built from components developed using a number of domain-specific languages, which have poor tool support, lack support to create reusable user interfaces, do not support binding data to user interfaces, do not easily support hierarchical navigation and use asynchronous APIs in JavaScript. In this subsection we will go into each of these issues in more detail.

Polyglot programming. Web development involves a mix of domain-specific languages, such as HTML for creating user interfaces, CSS for styling, JavaScript for application logic, SQL for database querying and caching manifests for application caching. While the use of domain-specific languages support separation of concerns, their loose coupling makes early error detection (such as in-editor error highlighting) complicated.

Tool support. Tool support for mobile web development is sub-optimal. Due to the aforementioned loose coupling of web languages, very little cross-language support is offered by IDEs. The dynamic nature of the web languages prevents accurate implementation of typical IDE features such as code completion and reference resolving.

Continuation-passing style. Browsers force programs to be written in *continuation-passing style*. JavaScript in the browser runs on a single thread that is shared with the page renderer. Therefore, JavaScript calls that take a long time to complete can freeze the browser. As Javascript does not allow developers to create threads, many JavaScript APIs are defined as *asynchronous* APIs. Asynchronous computations are computed on a separate thread (managed by the browser), and call back to the Javascript thread when the computation completes. While synchronous calls return the result of their computation as a return value, asynchronous methods are passed a *callback function* (or *continuation*), which is called with the result when the computation has finished. For instance, to obtain the GPS location of the device, a call is made to the `getCurrentPosition` method, with a callback function with `pos` (the result of the `getCurrentPosition` call) as an argument. The callback function is called when the device's GPS location has been established. While asynchronous APIs have favorable performance characteristics, continuation-passing style leads to verbose, difficult to read and maintain code.

User interface reuse. While HTML was designed to support declarative definition of web interfaces, it lacks abstraction mechanisms to *reuse* fragments of HTML, e.g. to reuse a calendar widget or a gridview control. It also lacks support to *define* such reusable components. Some JavaScript frameworks, such as jQTouch[3] and jQuery Mobile[4] attempt to fix the *reuse* issue by inventing an encoding, for instance, by interpreting the class attribute of tags as UI control encodings. For example, `<div class="calendar"/>` might encode a calendar control and is replaced dynamically with an appropriate calendar widget when the page is loaded. Nevertheless, such mechanisms only allow *use* of controls built into the framework, while *definition* of new controls has to be done using non-declarative JavaScript. Other frameworks, such as GWT[5] and Sencha Touch[6] replace HTML altogether with a Java (GWT) or JavaScript (Sencha) API to imperatively construct user interfaces.

[3] `http://jqtouch.com`
[4] `http://www.jquerymobile.com`
[5] `http://code.google.com/webtoolkit/`
[6] `http://www.sencha.com/products/touch/`

Data binding. While user interfaces are used to present and manipulate data, establishing a connection between data and the user interface requires a lot of boilerplate code. Data values have to be copied into the user interface when it is first loaded and stored back into data objects when certain events occur (e.g. when a "Save" button is pushed). Changes to data often give rise to changes in the user interface. When the user of a todo application creates a new task in the database, the screen that displays all tasks has to be updated. In many frameworks this behavior is not automatic and has to be implemented imperatively.

Navigation. The web is navigated by clicking hyperlinks, sending the user from one web page to another. Browsing patterns can be random, and websites are not always organized in a strictly hierarchical manner. We observe that in mobile applications, navigation patterns are more stringent. Data-driven mobile applications typically organize information as trees. Some applications present the top-level of the tree as *tabs*, enabling the user to quickly switch between them. Deeper levels of information are presented in *list views*. When the user selects a list item, the current screen slides to the left, and a new one slides in from the right. Navigation between screens usually happens by navigating deeper into the hierarchy or moving back to a higher level (using the *back* button). Moving between siblings of the tree is only supported by some applications, specifically news readers that allow the user to go to the previous or next article.

In application frameworks, browsing through the information tree creates a stack of screens where only the top of the stack is visible. When an item is selected, a new screen, representing the item is pushed onto the stack and when the user pushes the back button, the screen at the top is popped off the stack and the previous screen appears. This screen stack has to be managed *manually* by the developer, by pushing and popping screens.

3. MOBL BY EXAMPLE

To solve the problems outlined in the previous section, we have developed mobl, a new declarative domain-specific language supporting the development of mobile web applications. Mobl is supported by an Eclipse IDE plug-in, which provides in-line error highlighting, code completion, code folding and an outline view. Control-clicking the name of a variable, control or function call, navigates to its definition, which is invaluable for understanding larger projects. Figure 7 shows the mobl IDE in action. Mobl programs can also be compiled with a command-line compiler, which is useful in particular for continuous integration.

This section gives a tour of the language illustrated with the implementation of *ConfPlan*. In the next section, we discuss the language design features that enable the definition of ConfPlan as laid out in this section.

3.1 ConfPlan

ConfPlan is a mobile application that aids conference attendees in organizing and exploring the schedule of the conference they are attending. The conference schedules are easily browsable, searchable and events can be marked as favorite. Figure 1 through Figure 6 give an impression of the ConfPlan application in action. When the application is loaded first, it determines the user's geographical location and sends a request to the server that fetches conferences in

Figure 1: Nearby conferences

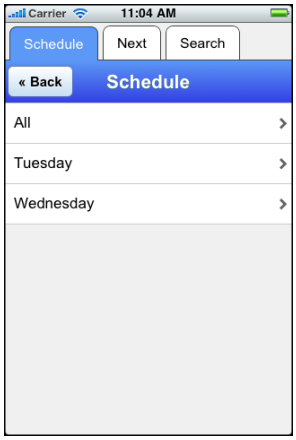

Figure 2: Schedule days

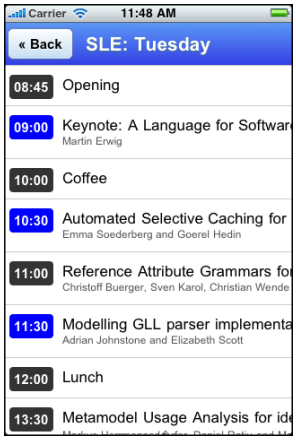

Figure 3: Tuesday's schedule

Figure 4: Talk details

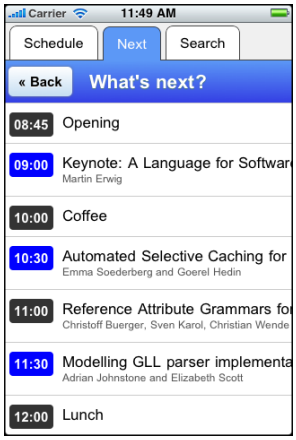

Figure 5: Next talks

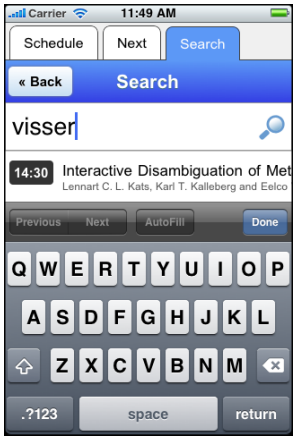

Figure 6: Search

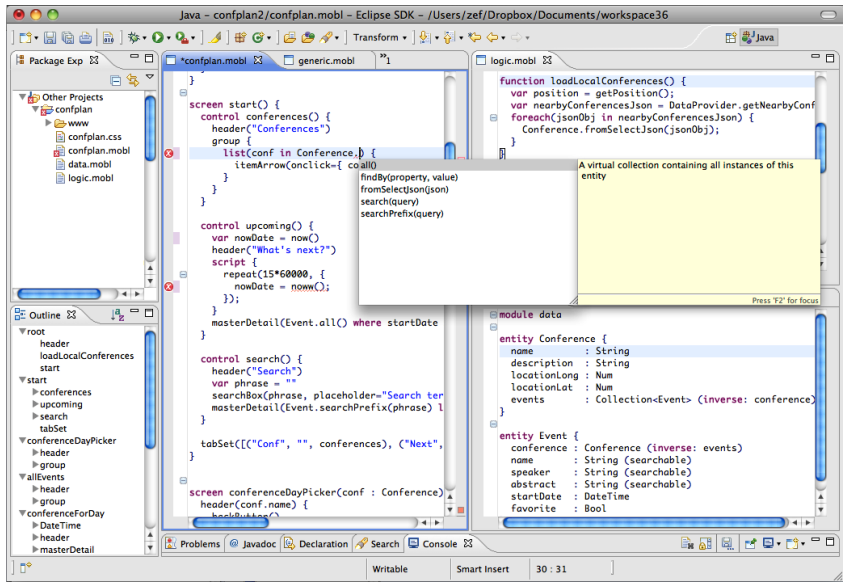

Figure 7: The Mobl Eclipse IDE

```
entity Conference {
  name          : String
  description   : String
  locationLong  : Num
  locationLat   : Num
  events        : Collection<Event>
                  (inverse: conference)
}

entity Event {
  conference : Conference
               (inverse: events)
  name       : String  (searchable)
  speaker    : String  (searchable)
  abstract   : String  (searchable)
  startDate  : DateTime
  favorite   : Bool
}
```

Figure 8: ConfPlan's data model

the vicinity. The user is presented with a list (Figure 1), and when he or she picks one, the conference's entire schedule is fetched from the server and cached locally on the device (if it was not cached before).

Subsequently, the user is presented with a tabbed view that presents three ways of exploring the schedule, either by day (Figure 2), by what is going to be presented next, based on the current time (Figure 5), or by searching the schedule (Figure 6). Conference events show the time they start (in blue when marked as favorite – tapping the time toggles it as a favorite), the title of the event and the speakers. When an event is tapped, a detail screen gives more details (Figure 4), including the talk's abstract. Searching through events using the "Search" tab (Figure 6) is instantaneous, the list of results updates as the user types.

3.2 ConfPlan Implementation

Figure 8 shows the declaration of ConfPlan's data model. It consists of two persistent data *entities*: `Conference` and `Event`. A conference has a name, description, location and a collection of events (such as talks and coffee breaks). An event belongs to a conference (and defines an inverse relationship with a conference's `events` property), has a name, speaker, abstract, start date and can be marked as a favorite. For each property a type is specified and optionally one or more annotations. The `searchable` annotations makes properties searchable (as can be seen in the `search` control in Figure 10).

Figure 9 details the startup sequence of ConfPlan. When a mobl application starts, the `root` screen is first loaded. The root screen displays the text "Fetching data..." and executes a script that first invokes the `loadLocalConferences` function, after which the `pickConference` screen is called, navigating the user to a screen where he or she can pick a conference (Figure 1).

```
screen root() {
  "Fetching data..."

  script {
    loadLocalConferences();
    pickConference();
  }
}
service DataProvider {
  resource getNearbyConferences(lat : Num,
             long : Num) : [JSON] {
    uri = "/nearbyConferences"
  }
  resource fetchProgram(conferenceId : String)
             : [JSON] {
    uri = "/conferenceProgram"
  }
}
function loadLocalConferences() {
  if(isOnline()) {
    var pos = getPosition();
    var nearbyConferencesJson =
      DataProvider.getNearbyConferences(pos.lat,
                                        pos.long);
    foreach(jsonObj in nearbyConferencesJson) {
      Conference.fromSelectJson(jsonObj);
    }
  }
}
screen pickConference() {
  header("Conferences")
  group {
    list(conf in Conference.all()) {
      item(onclick={ conference(conf); }) {
        label(conf.name)
      }
    }
  }
}
```

Figure 9: Loading conference data and picking a conference

The `loadLocalConferences` functions checks whether the user currently has access to the Internet, and if so, obtains

the user's location. After the location is known, a request is sent to the `DataProvider` service. A service definition specifies an interface to web services located on the server. `DataProvider` specifies two `resource`s, representing RESTful [12] web services. Resources can have arguments and a return type. The two resources of `DataProvider` both return JSON[7] objects. A JSON object is textual, structured data representation that can be imported into the database on the device using the `fromSelectJson` method defined on entities.

When the conference objects are loaded, or no Internet connection is available (in which case it is assumed there is cached data available in the database already), the `pickConference` screen loads (Figure 1). This screen is composed of a number of user interface *controls*. Mobl comes with a number of reusable control libraries, but developers can also easily define their own controls. Controls can have (named) arguments (such as `header`), and a body (such as `group`). The `pickConference` screen displays a header, and a group of items. `list` is a built-in construct that iterates over a collection and renders an item (specified in the `list` body) for each object in the collection. A `list` automatically responds to changes in the collection it iterates over. Consequently, when a new conference is imported into the database, or any of the conference data changes, the list will automatically be updated to reflect these changes. When an item in the list is tapped, the user navigates to the `conference` screen. The `onclick` argument of the `item` control is an example of a named argument with a *callback* as value. A callback is a snippet of script (enclosed in curly braces) that is executed when a certain event occurs (in this case a click or tap).

Figure 10 details the implementation of the tabbed `conference` screen (as can be see in Figure 2, Figure 5 and Figure 6). When loaded, a call to the `fetchConferenceSchedule` function fetches the schedule from the server if it is not already present. Furthermore, it defines three *controls*, one for each tab. The screen uses the `tabSet` control to display the three controls as a tab set. The `tabSet` control is passed an array of tuples, each with the name of the tab, and the control that is used to render it. `tabSet` is an example of a *higher-order* control, a control that takes other controls as arguments.

Both the `next` and `search` controls use the `masterDetail` control, which takes three arguments: a collection of data items, a control to be used to display in a list view, and a control to render the detail view. The result can be seen in Figure 5, Figure 4 (the `eventDetails` view) and Figure 6. The implementation of `eventItem` and `eventDetail` is shown in Figure 11. The `eventItem` control uses a `floatBox` control to display the start time of the event on the left (5 pixels from the left), the time blob is styled depending on whether the event is marked as a favorite or not. Whenever the time is tapped, the value of its favorite property if flipped, and the style of the time blob automatically updates. The collections passed to the master-detail control in `next` and `search` are using *collection filters*, which can be used to filter, order and limit the length of a collection using a syntax similar to SQL.

[7]JavaScript Object Notation `http://json.org`

```
screen conference(conf : Conference) {
  script {
    fetchConferenceSchedule(conf);
  }
  control schedule() {
    header("Schedule") { backButton() }
    group {
      item(onclick={ conferenceAll(conf); }) {
        "All"
      }
      list(startDate in determineDays(conf)) {
        item(onclick={
          conferenceForDay(conf, startDate);
        }) {
          label(daysOfWeek[startDate.getDay()])
        }
      }
    }
  }
  control next() {
    var nowDate = now()
    header("What's next?") { backButton() }
    masterDetail(conf.events
                   where startDate > nowDate
                   order by startDate asc limit 10,
                 eventItem, eventDetails)
    script {
      repeat(15*60000, {
        nowDate = now();
      });
    }
  }
  control search() {
    var searchPhrase = ""
    header("Search") { backButton() }
    searchBox(searchPhrase, placeholder="Phrase")
    masterDetail(Event.search(searchPhrase)
                   where conference == conf limit 10,
                 eventItem, eventDetails)
  }
  tabSet([("Schedule", schedule), ("Next", next),
          ("Search", search)])
}
```

Figure 10: The conference screen

The master-detail control of the search tab in Figure 10 uses the `Event.search(searchPhrase)` collection to generate items, the `searchBox` is bound to that same `searchPhrase` variable, resulting in updates to `searchPhrase` whenever the search phrase is changed in the search box. These changes are propagated to the master-detail control, automatically updating the list of search results as the user types. Similarly, the filtered collection in the `next` control depends on the value of `nowDate`, which is updated every 15 minutes, automatically updating the list of upcoming events.

4. LANGUAGE DESIGN DECISIONS

The previous section demonstrated how mobl is used to build useful applications with a small amount of code. This section analyses the design decisions behind the notation of the mobl language.

4.1 Integrated Language

In previous work [6], we have argued that the creation of a language that supports separation of concerns but is *linguistically integrated*, enables static consistency checking across concerns, catching many common programming errors early in the development process. Mobl applies this principle to the mobile web application domain, combining data modeling, user interface design, service description, and scripting into a single, statically typed language. The IDE checks

```
control eventItem(evt : Event) {
  floatBox(left=5) {
    block("timeBlob" + (evt.favorite ? "Fav" : ""),
      onclick={ evt.favorite = !evt.favorite; }) {
      label(buildTimeString(evt.startDate))
    }
  }
  block("event") {
    label(evt.name)
    block("speaker") {
      label(evt.speaker)
    }
  }
}
control eventDetails(evt : Event) {
  table {
    row {
      headerCol { "Title:" }
      col { label(evt.name) }
    }
    row {
      headerCol { "Speaker: " }
      col { label(evt.speaker) }
    }
    row {
      headerCol { "Start time: " }
      col { label(evt.startDate) }
    }
    row {
      headerCol { "Favorite: " }
      col { checkBox(evt.favorite,
                     label="Favorite") }
    }
  }
  block("textBlob") {
    label(evt.abstract)
  }
}
```

Figure 11: The `eventItem` and `eventDetails` controls

the application for inconsistencies, such as type errors, nonexistent properties, and controls with missing arguments, providing immediate feedback during development through inline error reports. Code completion helps the developer to explore the API and reference resolving lets the developer quickly jump to the definition of any screen, control, variable or service. Reuse of language elements, such as expressions, leads to a language with consistent syntax and semantics.

4.2 Automatic Data Persistence

Entity declarations such as shown in Figure 8, define a data model of *persistent* data. Mobl automatically creates a database on the mobile device and manages the database's schema. Database tables are created for each entity, as well as coupling tables where required.

Entity instances are automatically and transparently persisted to the database, without the developer having to write `INSERT` and `UPDATE` SQL queries. When a property in the data model is marked as `(searchable)`, a full-text index is created that can be queried using the entity's `search` method, as demonstrated in Figure 6. The search index is automatically updated as searchable properties change.

Rather than using unchecked SQL queries embedded in strings, queries in mobl are automatically derived from collection filters using `where`, `order by`, and `limit` clauses. The mobl compiler translates collection filters to efficient SQL queries. Because queries are integrated into the mobl language, their syntax is checked and it is ensured that properties referenced in the query exist and are of correct types. Query code completion helps the developer in selecting the right properties and values.

4.3 Reactive Programming

In spreadsheets, changing a single cell can cause a large number of other cells to update their values as well, because their value depends on the changed cell's value. The type of programming model exposed by spreadsheets is called *reactive programming* [5], a declarative style of programming where values are automatically recalculated when their dependent values change. Mobl's user interface definitions work similarly. The `search` control in Figure 10 defines a variable `searchPhrase`. It *binds* this variable to the `searchBox` control, creating a bi-directional connection between the two — when the value of the search box changes, it updates the variable and vice versa. Another control, the master-detail control, uses a collection as an argument that also depends on the `searchPhrase` variable. The collection is read-only, and therefore only a one-way connection from `searchPhrase` to the collection is established. Consequently, when the user modifies the text in the search box, the change is propagated to `searchPhrase`, in turn resulting in a change of the search collection passed to the master-detail control, immediately reflected in the list of search results.

Mobl uses the observer pattern [4] combined with reference arguments to realize this reactive programming model. Controls subscribe to change events of data values and update accordingly when events are triggered. Controls have *reference* arguments, instead of *value* arguments. Changes made to reference arguments, in contrast to value arguments, change the caller's actual argument as well. For example, consider a control `c` that takes an argument `x`. The control assigns the value `7` to argument `x`. After `c` is used on variable `v`, `v` would remain unchanged when arguments were passed as value. However, when passed as *reference*, as is the case in mobl, `v` would have value `7`. Therefore, typing text in the search box in Figure 10, the changed text value is assigned to `searchPhrase`.

4.4 Navigation

Section 2 described that navigation in mobile applications is hierarchical. Rather than managing the stack of screens manually, screen stacks in mobl are managed automatically. Screens are modelled as *functions* that can be called and optionally return a result. Like a function call, a *screen call* pushes a screen onto the stack and automatically pops it off the stack when a `screen return` statement is executed, e.g. when the user taps the back button.

The `backButton` control as used in Figure 10, has an optional `onclick` argument, similar to other types of buttons. A `onclick` callback of a back button is the most common place to perform a `screen return`. It is so common that the default value of the `onclick` callback argument is defined as a single screen return statement (as can be seen in Figure 12), and can therefore be omitted when called, as done in Figure 10.

Figure 13 shows an application of a screen with a return type. The `prompt` screen can be used to ask the user a question, and returns the answer as result. One or more

```
control backButton(text : String = "Back",
          onclick : Callback = { screen return; }) {
  <span class="backButton" onclick=onclick
        databind=text/>
}
```

Figure 12: Implementation of the back button

```
screen prompt(question : String) : String {
  var answer = ""
  header(question) {
    button("Ok", onclick={
      screen return answer;
    })
  }
  group {
    item { textField(answer) }
  }
}
...
alert("Hello, " + prompt("First name?") +
      " " + prompt("Last name?"));
```

Figure 13: The prompt screen

invocations of this screen can be combined in a single expression. The statement at the bottom of Figure 13 results in two prompt screens appearing in sequence. The first asks for the user's first name, the second for the last name. Subsequently, an alert dialogue pops up that greets the user with his or her full name.

4.5 Extensibility

An important goal of the design of mobl is extensibility. Existing frameworks and languages for mobile development provide a standard set of controls that are easy to use, while defining custom controls is much more complicated. Mobl, as a language, does not come with any controls built-in. All its controls are provided in mobl libraries, written in mobl itself, by composing existing controls, or using plain HTML or JavaScript at the lowest level.

HTML. Figure 12 shows the implementation of the `backButton` control, it uses a HTML `<span>` tag that is styled with CSS to look like a button. In mobl, HTML attribute values are *mobl expressions*, and can thus perform arbitrary computations. The special `databind` attribute binds a variable to the *value* of the tag. In the case of a `<span>`, *value* is defined as its body, for `<input>` tags, it is defined as the text in the control.

JavaScript. In addition to entities, which are automatically persisted in the database, mobl also supports regular, non-entity types. Examples of these include strings, numbers, arrays, GPS locations and dates. JavaScript has these types built in, but in order to expose them to mobl applications their interfaces need to be specified. This is done using *external definitions*. Figure 14 shows a fragment of the definition of the generic `Array` type. Similar to Java, mobl supports generic types. External types only define the interface of a type. The implementation is provided by a library, or, as is the case with `Array`, by the JavaScript runtime. The `sync` keyword used for functions indicates that a function is *synchronous*, i.e. it returns a value immediately, rather than taking a callback function that will be called with the result.

```
external type Array<T> {
  length : Num
  sync function push(item : T) : void
  sync function join(sep : String) : String
  sync function contains(el : T) : Bool
  sync function insert(idx : Num, item : T) : void
  sync function remove(item : T) : void
}
```

Figure 14: A fragment of the definition of `Array`

```
getCurrentPosition(function(pos) {
  DataProvider.getNearbyConferences(pos.lat,pos.long,
    function(nearbyConferencesJson) {
      for(...) {
        ...
      }
    });
});
```

Figure 15: Asynchronous calls in JavaScript

External types are not the only type of external definition that mobl supports. Every type of definition has an external version, including entities, screen, controls, functions and services. Therefore, it is possible to create a custom JavaScript implementation of any of these definitions and expose them through mobl. This has enabled us to reuse controls provided by other mobile frameworks and use them as mobl controls.

4.6 CPS Transform

Numerous JavaScript APIs, such as geolocation and AJAX, are asynchronous; instead of calling a method and return a value immediately, they are called with a *callback function* that is invoked when the result of the computation is known. This can be milliseconds up to many seconds later, depending on the computation. JavaScript supports anonymous functions as expressions, which makes calling asynchronous APIs less verbose. The `loadLocalConferences` function in Figure 9 performs two asynchronous calls: one to determine the user's current location and another to a web service that retrieves nearby conferences. In JavaScript, calling these asynchronous methods in sequence looks as shown in Figure 15. The `getCurrentPosition` function is passed a function with an argument: `pos`. When the user's location is known, this callback function is called with the location as argument. The function calls another asynchronous method `getNearbyConferences` of the `DataProvider` object. It passes the latitude, longitude and another callback function to that method. When the list of conferences is retrieved, the callback function is called. The function subsequently processes and imports the retrieved data into the local database. Compared to Figure 9, this continuation-passing style is clearly harder to read and more verbose.

To avoid having to write code such as Figure 15, mobl exposes asynchronous APIs as regular synchronous APIs. The compiler uses *continuation-passing style (CPS) transformation* [13] to *transform* code that calls asynchronous APIs automatically to the continuation-passing style as shown in Figure 15.

5. DISCUSSION

We have constructed a number of applications using mobl, ranging from simple toy applications such as a todo list manager and a tip calculator to more complex applications such as a twitter client, the ConfPlan application and even simple graphical games and a collaborative drawing applications[8]. While developing these application we grew a library of reusable controls, ranging from basic, such as labels and buttons, to more complex, such as the tab set, a master-detail, accordion, date picker and context menu controls. The definitions of these controls are all declarative and concise. Mobl's screen calling mechanism leads to clean code and a navigation style that feels native to the application user.

This section discusses the limitations of our approach and compares it to related work.

Language Limitations. While mobl's type checker checks many program properties, it does not yet check everything. For instance, certain controls have to be nested within other controls. For instance, `item`s need to be nested inside `group`s to be rendered properly. Mobl does not yet support declaring such nesting requirements.

While mobl integrates languages for many application aspects, it does not yet integrate a styling language. Currently, CSS styles are defined separately from the rest of the application, lacking cross-language checking or IDE support. Integrating styling into mobl is future work.

Mobile web applications generated by mobl are portable to any mobile platform that supports HTML 5. However, the user interface does not adapt to the look-and-feel of the platform, nor does it adapt to the screen size of the device. We intend to extend controls with the ability to better adapt to their environment. For instance, controls such as the master-detail control are sufficiently high-level to allow an implementation specific to tablet-sized screens that take advantage of the wider screen by rendering the list of items on the left and the detail view on the right.

Performance. The performance of mobile *web* applications will always be worse than native applications, just as web applications in general are slower than native desktop applications. Nevertheless, by caching both the application and its data locally and the recent performance improvements of (mobile) browsers, performance of mobile web applications is very reasonable. While performance has not been a primary focus of the mobl compiler thus far, there is potential to generate more optimal code, both in size (by e.g. removing white space from the output and reducing variable and method name size) and execution speed.

While load-time is a common performance bottleneck in web applications, mobl applications are cached locally on the device. A first load of ConfPlan over a 3G network takes about 10 seconds. Subsequent launches occur from the browser cache and are almost instantaneous.

[8] `http://github.com/zefhemel/mobl/tree/master/samples/`

Good Web Citizenship. While mobl uses the web as a medium to deliver applications, and uses web technologies to run applications, a mobl application is not built like a regular web application: a mobl application does not consist of pages with unique URLs; breaks the browser's back button; and is not indexable by search engines. We intend to solve some of these issues. A working back button is relatively easy to implement. Full history support is much more complex, requiring some type of encoding of the application state in the URL of the application. Indexing mobile applications can be useful for some data-driven applications. A tool such as CrawlJax [10] could be used to generate a static, indexable version of the application.

Web Application Limitations. While web applications have the advantage of being portable, they have limitations too. HTML5 offers many JavaScript APIs that give access to various device services, but their implementation in mobile devices is not always complete. Access to audio and video services is limited — it is possible to play an audio or video file, but only by launching the dedicated audio or video player. Access to other device-specific features such as bluetooth, the built-in compass, camera and local file storage are not supported yet.

A way around these restrictions is a native/web hybrid approach. PhoneGap[9] allows a developer to build applications using web technologies, and expose additional native APIs including a file storage API and a camera API through JavaScript, an approach that works nicely with mobl. Applications built with PhoneGap can be deployed as native applications through e.g. the Apple AppStore or Android Marketplace.

Web applications have limitations in user experience as well. It is very difficult to reproduce certain native application behaviors in web applications. Inertia scrolling is one such behavior, where, after a finger flick on the screen, the screen keeps scrolling for a while longer after the finger no longer touches the screen. There are a number of projects that attempt to emulate this behavior in the browser, but it has proven very difficult to do perfectly. Fixed positioning is another behavior that is difficult to achieve in mobile browsers. A control that has a fixed position, does not move when the rest of the screen scrolls. A typical example is a screen header. A header is positioned at the top of the screen and while the rest of the content scrolls, the header remains fixed at the top.

5.1 Related work

WebDSL. In previous work we developed WebDSL [14], a domain-specific language for the development of RESTful web applications. WebDSL applications are stateless — no application state is maintained on the server. The only state that is maintained is database state. In contrast, mobl relies heavily on application state. For instance, for the screen stack and local variables in controls. Navigation through a WebDSL web application is also different than mobl. WebDSL defines pages with links between them, while navigation in mobl applications is purely hierarchical. Simi-

[9] `http://www.phonegap.com`

lar to mobl, WebDSL is a statically typed language enabling static verification of web applications [6]. WebDSL and mobl generate very different types of web applications. WebDSL generates Java source code that runs on the server, while mobl generates JavaScript code that runs on the client. Instead of controls, WebDSL pages are composed of templates. While WebDSL developers can define their own templates, core controls, such as labels, inputs and buttons are built into the compiler and cannot easily be defined by the developer.

DSLs for mobile development. Behrens [1] describes a domain-specific language for creating *native* mobile applications, using a single language from which both iPhone and Android applications can be generated. Similar to mobl, the language comes with an IDE plug-in for Eclipse that supports error high-lighting, code completion and reference resolving. Berhens' language has a number of high-level controls built into the language, including sections, detail views and cells. It can fetch its data from data providers. However, the DSL currently only supports data *viewing* and is not as flexible as mobl; defining custom controls is not supported, for instance.

Kejriwal and Bedekar developed MobiDSL [7], an XML-based language for developing mobile web applications. Unlike mobl, the application is executed on the server and plain HTML is sent to the mobile device. MobiDSL comes with a number of built-in controls, such as query views, page headers and search requests that can be used to build pages. It is not possible to define custom controls, nor is there specific IDE support available.

Google Web Toolkit is a tool that enables client-side web applications using Java. The use of Java has the advantage of having excellent IDE support. A GWT plug-in[10] enables access to HTML 5 APIs such as geolocation and local databases. Like mobl, GWT applications are compiled to a combination of HTML, Javascript and CSS. Defining user interfaces using GWT is not very declarative, however. A Java Swing-like API is used to imperatively create user interfaces.

Functional Reactive User Interfaces. Courtney and Elliot developed Fruit [2], a Haskell framework that applies functional reactive programming [11, 3, 15] to user interfaces. It is based on signals (streams of events) and signal transformers (functions that transform streams of events). On top of these concepts, Fruit builds a purely functional user interface library. Mobl's user interfaces are also reactive, but not based on pure functions. Concepts such as signals and signal transformers are not exposed to the developer in mobl. Instead, events triggered by changes in data or control events, result in updates to the user interface.

6. CONCLUSION

In this paper we have introduced mobl, a new language for developing mobile web applications. Mobl integrates languages for data model definitions, user interface, web services and scripting. Mobl is high-level since it supports the definition of reusable controls, as demonstrated by the extensive library of controls, all defined using mobl. Mobl is extensible since it supports use of HTML and Javascript. It is declarative since its user interface are automatically updated through reactive programming, data is automatically persisted in the client-side database, data retrieved from web services is automatically imported into the database. We built a number of applications using mobl, including task list managers, twitter clients and ConfPlan. In the future we intend to work on integrating a styling language similar to CSS and add mechanisms to adapt user interfaces to the native platform look-and-feel and take advantage of available screen estate.

[10] `http://code.google.com/p/gwt-mobile-webkit/`

7. ACKNOWLEDGMENTS

This research was supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution*. We would like to thank Google for providing Android phones for testing and development.

8. REFERENCES

[1] H. Behrens. MDSD for the iPhone. In *SPLASH '10: Proceedings of Object oriented programming systems languages and applications companion*, pages 123–128, 2010.

[2] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *PLI*, pages 41–69, 2001.

[3] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP*, pages 263–273, 1997.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[5] D. Harel and A. Pnueli. On the development of reactive systems. *Logics and models of concurrent systems*, page 477, 1985.

[6] Z. Hemel, D. Groenewegen, L. C. L. Kats, and E. Visser. Static consistency checking of web applications with webdsl. *JSC*, 2010.

[7] A. A. Kejriwal and M. Bedekar. MobiDSL - a domain specific langauge for mobile web applications: developing applications for mobile platform without web programming. In *Proceedings of the 9th OOPSLA Workshop on Domain Specific Modelling (DSM'09)*, October 2009.

[8] J. Kim, R. A. Baratto, and J. Nieh. pthinc: a thin-client architecture for mobile wireless web. In *WWW*, pages 143–152, 2006.

[9] A. M. Lai, J. Nieh, B. Bohra, V. Nandikonda, A. P. Surana, and S. Varshneya. Improving web browsing performance on wireless pdas using thin-client computing. In *WWW*, pages 143–154, 2004.

[10] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling ajax by inferring user interface state changes. In *ICWE*, pages 122–134, 2008.

[11] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64, 2002.

[12] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, May 2007.

[13] G. J. Sussman and G. L. S. Jr. Scheme: An interpreter for extended lambda calculus. AI Memos 349, MIT AI Lab, 1975.

[14] E. Visser. WebDSL: A case study in domain-specific language engineering. In *GTTSE*, pages 291–373, 2007.

[15] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *PLDI*, pages 242–252, 2000.

TUD-SERG-2011-001
ISSN 1872-5392